INTEGRATING PLANNING, EXECUTION, AND LEARNING

Daniel R. Kuokka

Computer Science Department[1]
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

To achieve the goal of building an autonomous agent, the usually disjoint capabilities of planning, execution, and learning must be used together. This paper describes an architecture, called MAX, within which cognitive capabilities can be purposefully and intelligently integrated. The architecture supports the codification of capabilities as explicit knowledge that can be reasoned about. In addition, specific problem solving, learning, and integration knowledge is developed.

1. Introduction

The expense, isolation, danger, and uncertainty involved in space research vividly points out the need for robust autonomous robots. However, the current generation of intelligent systems only present a subset of the requisite behavior. For an intelligent system to be a competent autonomous agent, many different cognitive capabilities must be solidly integrated. For example, an autonomous system is more than a planner that generates an answer given a precise problem specification; it must actually take actions dictated by its reasoning. However, it is not sufficient merely to give a plan to a simple execution system; the unpredictability of the world makes it very unlikely that the plan is correct or complete. A system must be able to suspend planning or execution in order to seek needed information. However, the acquisition of knowledge, itself, may require planning and execution. Thus, there is a complete interdependence between the various cognitive capabilities.

These issues stem from the relaxation of two assumptions commonly made in problem solving: the availability of complete knowledge, and a static environment. Complete knowledge is fundamental to the plan-then-execute paradigm assumed by many problem solvers. However, it is an assumption that is simply invalid in many cases. The ability to take action based on incomplete information, and to intentionally acquire new knowledge, is fundamental if a system is to operate robustly in real environments. Another complication is that real-world environments tend to be dynamic. A successful agent must always be aware of the external world, and it must be able to suspend any task in favor of a more urgent task. The problem of integrating planning and execution in a dynamic world is even more complex if consideration is given to the fact that reasoning, itself, is not a "free" activity. Reasoning requires time, and it is not always safe to assume that a system has as much time to think as required. A successful autonomous agent must

[1]This research was supported in part by ONR grants N00014-79-C-0661 and N0014-82-C-50767, DARPA contract number F33615-84-K-1520, NASA contract number NCC 2-463, and a grant from the Hughes Aircraft Corporation. The views and conclusions contained in this document are those of the author alone and should not be interpreted as representing the official policies, either expressed or implied, of the US government, ONR, DARPA, NASA, or the Hughes Aircraft Corporation.

be able to rationally control its reasoning, just as it must rationally control its physical capabilities.

This paper describes work in progress aimed at *integrating* previously separate capabilities into a unified autonomous system that can function in an unknown and dynamic environment. The focus is the synergistic interaction of the components, rather than better individual components. There are two main components of the research. First is the development of an architecture, called MAX, that permits the principled, knowledge-based integration of complex capabilities via meta-level reasoning. This is discussed in section 3. Second is the codification of knowledge that intelligently interleaves each capability. This is presented in section 4. Section 5 describes an extended example in which all parts of the system work together. However, before delving into MAX, a brief survey of other work is presented.

2. Related Work

Prior research has tended to focus on various subproblems of building autonomous agents, such as planning, learning, and vision, under the principle of divide and conquer. There has been little work on systems that integrate such behavior. However, there has recently been an increase of interest in the high-level control of autonomous agents, as well as cognitive architectures that cover, in principle, a broad range of capabilities. These systems can be roughly organized according to their reliance on explicit reasoning.

At one end of the spectrum are those systems, such as Pengi [1], that propose situated activity as the mechanism behind higher level planning. The research of Brooks [3] emphasizes the use of a hierarchy of behavioral components, each of which uses perceptual information directly to produce some behavior. Kaelbling [8] also tends toward such an architecture, but she also suggests the need for explicit planning. Another step toward explicit reasoning is represented by the Procedural Reasoning System system of Georgeff et al [6] which does pattern-directed invocation of canned procedures. These systems posit, to varying degrees, that high level planning is not required, as it can emerge from lower-level behavior. This thesis has not yet been adequately proven.

Moving away from robot control architectures, a number of systems propose a relatively weak integration of intelligent modules. This model is exemplified by plan-then-execute systems such as NOAH [13] and SIPE [16]. Another form of loose integration is the black-board architecture, used by systems such as BB1 [7] and Codger [15]. In general, the top-level control structures used by these loosely integrated systems only address a subset of the integrated behavior required of autonomous agents.

Strongly integrated systems are represented by the set of general architectures intended to be applicable to any task. Cognitive architectures such as ACT* [2] and SOAR [9] have achieved a fair degree of success in this regard. However, they have not been applied to the task of rationally integrating various intelligent capabilities in a dynamic environment. Furthermore, the assumptions imposed by the added constraint of cognitive plausibility may make such integration difficult. Another candidate architecture is PRODIGY [11], which provides a variety of base-level reasoning and learning mechanisms. In fact, the work described in this paper is an effort to evolve PRODIGY into a competent meta-level reasoner.

Further along the spectrum are systems designed explicitly for meta-level reasoning. Examples include MOLGEN [14], MRS [5], and Theo [12]. These systems are capable of very elaborate reasoning, but their application to a real-time, irreversible environment is problematical.

Since there hasn't been much work on meta-level architectures intended for autonomous agent control,

there is very little in the way of meta-level knowledge required by such a system. Only pieces have been investigated. The work on execution monitoring includes some techniques for coping with unexpected situations, but only in restricted environments. Carbonell and Gil [4] have developed techniques to guide experimentation to learn operators, and Eurisko [10] also performs some experimentation. However, the higher level issues of intelligently integrating experimentation into an autonomous agent remain unanswered.

3. The MAX Architecture

The main goal of the MAX architecture is to allow cognitive capabilities (e.g., planning and experimentation) to be flexibly controlled. A powerful way to accomplish this is to enable the system to reason about mental actions (perform meta-level reasoning) as well as reason about physical actions (perform base-level reasoning). This, in turn, requires that system's cognitive capabilities be encoded as explicit knowledge. It is this approach that underlies the design of MAX.

To elaborate, consider a traditional system in which the capabilities are hardwired into the kernel; the capabilities can be applied only according to the foresight of the designer and the flexibility of the algorithm. Undoubtedly, unforeseen situations will crop up with which the system cannot cope. Now consider a system in which its capabilities are explicit knowledge. Such a system can use its entire reasoning power to flexibly apply its capabilities. The system is no longer restricted to some hardwired interaction between planning, execution, and learning. Furthermore, since the capabilities are encoded just like any base-level knowledge, their implementation can use arbitrary amounts of knowledge to obtain expert performance, and the capabilities can be modified by the system.

Encoding cognitive capabilities as knowledge requires that the knowledge representation be sufficiently powerful, and that the system have the ability to execute explicit knowledge. It is these two requirements that the MAX architecture is designed to meet. Section 3.1 describes MAX's knowledge representation, which is powerful enough to implement cognitive capabilities while remaining simple enough to reason about. Next, section 3.2 describes the control structure, which allows explicit knowledge to be executed.

3.1. Knowledge Representation

The foundation of MAX's knowledge representation is first order predicate logic augmented with a single data structure, called an *lframe* (for logic-frame). An lframe can best be described as an independent, explicit logical database, or state. Therefore, in contrast to traditional logic, where terms are atomic symbols only, terms in MAX can also be structures representing entire states. Since a state is a collection of logical assertions, an lframe is a recursive data structure, not unlike frames. However, there are two features of lframes that produce the real leverage of the representation: the capability to represent with one lframe an entire sets of states, and the presence of a set of high-level relations over lframes.

Consider figure 3-1 which shows a single lframe representing the definition of a Strips-style operator. Brackets denote lframes, and parentheses denote logical assertions, or literals. Within the operator, there are a number of assertions that specify its components: namely the input parameters, preconditions, adds, and deletes. Furthermore, each component is, itself, an lframe that represents a state (or complement of a state) of the blocks world. Note the **`$vars`** assertion, which declares a set of variables within the operator. The variables allow the lframe to represent an infinite number of operators, corresponding to the various instantiations of the variables. Also notice that the variables declared within the operator as a whole are referenced within the subcomponents of the operator. This constrains the values of subcomponents of the

operator that can co-occur, insuring the operator remains valid (i.e., if the precondition matches a particular object, the operator should not delete a different object from the state). The use of lexical variables is an important feature of lframes, allowing the succinct representation of an entire set of states.

```
[($vars ?block1 ?block2)
 (parameters [(topblock ?block1) (bottomblock ?block2)])
 (precondition [(holding ?block1)
                (clear ?block2)])
 (delete [(holding ?block1)
          (clear ?block2)])
 (add [(on ?block1 ?block2)])]
```

Figure 3-1: Definition of the put operator

The other main feature of lframes is the definition of a set of high-level relations, most notably **match**, **intersection**, **union**, and **difference**. These relations are basic to the reasoning process, just as **on** is basic to the blocks world. These basic operations allow reasoning to be represented naturally as explicit knowledge. Consider figure 3-2, which gives the definition of operator **apply**. This operator states the effects of applying a *subject* operator, for example, **put**. The preconditions decompose the subject operator into its components, verify that the preconditions match the current state, and specify the relationship between the old state, deletes, adds, and new state. The delete and add of **apply** then specify the changes to the state of the planner. The structure of the operator is identical to the base-level operator, only the relations have changed.

```
[($vars ?op ?del ?add ?state1 ?state0 ?state2 ?oldplan ?newplan)
 (parameters [(operator ?op)])
 (precondition [(state ?state1)
                (match ?op [($vars ?pre)
                            (precondition ?pre)
                            (match ?state1 ?pre)
                            (delete ?del)
                            (add ?add)])
                (difference ?state1 ?del ?state0)
                (union ?state0 ?add ?state2)
                (plan ?oldplan)
                (postpend ?oldplan ?op ?newplan)])
 (delete [(state ?state1)
          (plan ?oldplan)])
 (add [(state ?state2)
       (plan ?newplan)])]
```

Figure 3-2: Definition of the apply operator

In summary, lframes are vital to the MAX architecture for several reasons. Lframes are an appropriate data structure for representing the basic fodder of reasoning, namely states of knowledge. The capability to introduce variables within lframes allows complex sets of states to be succinctly represented. Such a capability is not present in either traditional logic or frame-based representations. The definition of basic relations over lframes allows the consequences of knowledge to be represented without resorting to representational details (i.e., traditional logic could be used to represent the apply operator, but the result would be the implementation of a theorem prover in logic.) These features make it practical to *represent* complex reasoning, a necessity in the rational integration of high-level capabilities.

3.2. Control Structure

In addition to the knowledge representation, the other main component of the MAX architecture is the control structure. The main goal is to allow the control of the agent to be based on explicit knowledge. This supports a very flexible flow of control, avoiding the problem of control being hardwired into the kernel. Indeed, there must still be some kernel that interprets the knowledge, but once the powerful capabilities are removed, the kernel can be very simple. In other words, the kernel of MAX is just powerful enough to *execute* knowledge rather than reason about knowledge.

The basic unit of the control structure is called a *task*, which is an Iframe structure. A task can be thought of as an explicit representation of a production system with two exceptions. First, the "productions" are divided into operators and control rules representing what can be done and when to do it, respectively. This allows a variety of control to be applied to a fixed set of actions (i.e., even though actions are determined by the physical capabilities of the agent, when to do them is flexible). Second, an operator (called a *complex* operator) can invoke a subtask that does an arbitrary amount of processing, similar to operator implementation problem spaces in SOAR. This allows an agent to reason about abstract actions, such as building a bridge or creating a plan that achieves a goal.

Consider the example of figure 3-3, which shows a planning task. The operators and control rules correspond to traditional productions. The state corresponds to working memory. In this example, the planning state consists of a blocks world goal, a current blocks world state, the plan being formed, and a domain theory specifying the legal operators for the blocks world. This illustrates how the same language can be used both to implement reasoning, and to represent knowledge (the object of reasoning). Finally, the **pending** assertion indicates that this task was invoked by executing a complex operator within a higher-level task, called **solve**. When the planning task is finished, the results are added to the solve task, which is then resumed.

```
[(rule apply-satisfied-operator [(condition [...])
                                 (action subgoal [...])])
 (rule subgoal-on-goal-state-diff [...])
 ...
 (operator apply [(precondition [...])
                  (delete [...])
                  (add [...])])
 (operator subgoal [...])
 ...
 (state [(goal [(on block1 block2)])
         (state [(on-table block1)
                 (on-table block2)])
         (plan [(elt 1 pick [...]) (elt 2 put [...])])
         (domain [(operator put [...])
                  (operator pick [...])])])
 (pending solve [(rule plan-when-difficult-goal [...])
                 ...
                 (operator plan [...])
                 ...
                 (state [...])])]
```

Figure 3-3: Task implementing a planner

The choice to build the MAX control structure on a production system was inspired by the successes encountered in the world of expert systems. Also, productions are a natural means of implementing a highly reactive system, a requirement for any autonomous agent. However, MAX is unique in that it

unifies production and working memory, in addition to process memory (tasks actually represent the execution state). This gives MAX extreme flexibility to reason about and modify its own behavior, which means nearly all control decisions can be considered explicitly. Finally, the separation of various capabilities into different tasks results in a modular system, a large advantage from the practical standpoint of system development.

4. Domain Knowledge

The MAX architecture provides a framework in which capabilities can be encoded and controlled. This section outlines the kind of capabilities that can be encoded. Each capability of the agent, from the most primitive (such as simple motion) to the very complex (such as planning), is encoded as a domain. A domain is defined as a set of operators and a set of control rules. Therefore, a task is a snapshot of the execution of a domain. Furthermore, since complex operators result in the execution of a subtask, there is a domain associated with each complex operator. Conversely, a complex operator represents the external specification of a domain.

Table 4-1 gives an overview of the key domains required to implement and integrate intelligent capabilities. Each row specifies the name of the domain (which corresponds to the complex operator that is used to invoke it), the key operators, selected control rules, and the main objects about which the domain reasons. Notice that many domains use complex operators corresponding to other domains; thus, the capabilities of the system are applied recursively.

The function of the select domain is to choose between various potential goals or activities. The select domain consists of two operators, solve a goal (described below) and run a heuristic procedure (which simply executes a specified domain). In addition, there are rules that specify when a particular operator should be applied. In general, goals concerning robot safety should be considered first, followed by primary goals, followed by background goals. Also, procedures should be used when available. However, there are likely to be interactions requiring specific exceptions. This illustrates the heuristic nature of the knowledge required in such domains; quite often there is no simple algorithm that provides the requisite behavior.

The solve domain provides a rather unique but powerful capability, namely that of intelligently selecting the problem solving approach used for a given goal. For example, the solve domain can choose between forming and executing a plan, specializing and executing a canned plan, and executing a heuristic procedure. The choice of problem solving tactic can be based on the presence of the requisite knowledge plus heuristics about the particular domain and problem. Finally, the solve domain includes the option of intentionally learning more about the environment or a domain theory. The importance of this is discussed below.

Within the plan domain, there are two main operators corresponding to the two main actions of means-ends analysis: subgoaling on an operator and applying an operator. Each of these operators affects the state of the planner by modifying the hypothetical state, the goal, or the partial plan. Notice that the apply operator is exactly that presented in figure 3-2. Unlike many problem solvers, a particular planning algorithm is not built in to the system. Alternate domains could be added that implement forward chaining or various forms of non-linear planning. Knowledge within a higher level domain would then be required to select the most appropriate technique.

The execute domain is, on the surface, rather simple. It simply takes a plan and attempts to execute it.

DOMAIN	OPERATORS	RULES	STATE
select	run	run-emergency-procedure	current-state
	solve	solve-given-goal	potential-goals
solve	plan	plan-when-difficult-goal	goal
	execute	execute-when-plan-exists	current-state
	run	run-heuristic-procedure	base-domains
	explore	explore-when-info-needed	
	experiment	experiment-when-faulty-ops	
plan	apply	apply-satisfied-operator	goal
	subgoal	subgoal-on-goal-state-diff	current-state
		exit-when-unsatisfied-axiom	base-domain
			plan-so-far
execute	step-plan	step-plan-when-as-expected	plan
	solve	subgoal-when-discrepancy	expected-state
		exit-when-large-discrepancy	perceived-state
explore	search-loc	search-inferred-location	needed-info
			current-model
			locs-searched
experiment	apply-op	apply-op-in-different-state	expected-state
	modify-op	modify-op-when-model-correct	perceived-state
			domain-theory
robot	move	pick-up-before-moving	robot-loc
	put		object-loc
	pick		

Table 4-1: An overview of cognitive domains

However, this entails more than blindly stepping through the plan; the expectations at each step must be examined and compared to the actual situation. If a significant discrepancy appears, the execute task must decide how to correct the problem. For instance, if the discrepancy is slight, it is probably best to patch the plan and continue. However, if the problem is significant, it is best to exit the execute task and allow the higher level problem solving task to address the problem. This is an important option because the partial execution of the plan may have resulted in the acquisition of knowledge that modifies the assumptions underlying the plan. The rules governing the decision to patch or abort a plan are another example of the heuristic knowledge required within many capabilities.

Even though a variety of failures can occur within the execute and plan domains, the basic cause is almost always incomplete or incorrect knowledge. For example, the execution of a plan can fail if the results of the operators are not as expected, and the planning domain can fail if key pieces of knowledge about the state are missing. A robust autonomous agent must be able to cope with such situations, motivating the need for an intentional knowledge acquisition capability. Two of the domains in table 4-1 supply such a capability. First, the experimentation domain implements the technique of Carbonell and Gil [4] in which the preconditions and effects of operators are learned by performing experiments. Second, the exploration domain allows the use of a partial knowledge structure in the process of augmenting that knowledge structure. For example, knowledge of a door to another room suggests that an agent should move through the door to discover the contents.

The ability to perform intentional learning requires more than the presence of learning techniques; the invocation of learning activities must be intelligently controlled. For example, an agent with an urgent task to accomplish should attempt all known approaches before embarking on a regime of experimentation to learn the best approach. Thus, the solve domain contains heuristics that control the application of the explore and experiment operators based on the current problem solving priorities. It is this level of control that is vital to the rational integration of learning and problem solving.

Finally, in addition to implementing the cognitive capabilities of the agent, domains are also used in their traditional role of representing the base-level capabilities of the agent. There is only one base-level domain listed in table 4-1, the robot domain. It is worthy of note that complex operators can be used within base-level domains, just as in cognitive domains, to introduce useful abstractions. For example, the *move* operator could invoke a subtask in which the details of path planning and obstacle avoidance are considered. This allows higher-level planners to reason about moving as if it were a simple atomic action.

5. A Day in the Life (of an Autonomous Agent)

The power of the MAX architecture derives from the intelligent integration of many relatively simple capabilities. To illustrate how the aggregate behavior comes about, this section presents an extended example that exercises many aspects of the system. The example is set within a simple household robot domain. For the purpose of this example, the sensory and effectory capabilities of the robot will be rather powerful. The robot has three main effectory capabilities, move, pickup, and putdown. These capabilities correspond to the operators within the robot domain outlined in the previous section. The robot has one main sensor that returns a map of the current room. Objects within the room are never obscured, but objects in other rooms cannot be seen. This provides limitations sufficient to introduce issues due to incomplete knowledge.

The top-level goal of the robot is to fetch a wrench. However, the tool is in another room, and the robot does not know its location. Furthermore, the robot has a number of standing goals such as preserving its own safety and coping with emergencies. The initial knowledge of the robot is limited to the condition of the room it currently occupies.

The top-level domain of the robot is the select domain. Within this domain, control rules select the solve operator on the goal of fetching the tool. This choice is made because there are currently no other unsatisfied goals. The robot enters the problem solving domain wherein the lack of any applicable plan or procedure causes the plan operator to be fired. This creates a subtask that actually does the planning by applying the subgoal and apply operators. Unfortunately, since the robot does not know where the wrench is, the planning task fails, and control returns to the solve task.

Within the solve domain, the failed plan operator asserts the reason for failure, namely that the location of the wrench is unknown. This causes a rule to fire that attempts to discover the needed knowledge via the explore operator. The explore operator sets up a new task in which the goal is to discover the location of the wrench. The exploration task uses the knowledge that doors lead to other rooms to drive its explorations. As the robot searches, it adds knowledge to its state model and it maintains a separate structure indicating where it has been. The mental state of the robot at this point in time, as represented by the current task, is shown in figure 5-1.

To introduce another dimension, assume that as the robot is searching for the wrench, it discovers a fire. Fires are known to be emergencies; therefore, a rule fires that suspends the explore task, allowing a higher-

level domain to address the problem. Similarly, the solve task suspends itself since it cannot cope with an emergency; it is only concerned with solving a particular goal. The select task is returned to, where emergency heuristics choose to deal with the fire instead of fetch the wrench. This causes the invocation of a different subtask containing the fire-fighting expertise.

```
[(rule search-inferred-location [...])
 ...
 (operator search-location [...])
 ...
 (state [(current-model [(at desk room1)
                         (at chair room1)
                         (at hutch room2)])
         (locations-searched [(elt room1) (elt room2)])])
 (pending solve
          [(rule plan-when-difficult-goal [...])
           ...
           (operator plan [...])
           ...
           (state [(goal [(at wrench room1)])
                   (planning-failure [(at wrench ?anywhere)])
                   ...])
           (pending select [...])])]
```

Figure 5-1: Mental state while exploring

Once the fire is extinguished, the conditions that originally caused the solve task to be applied to fetching the wrench are still present. Therefore, the robot resumes this activity, which reinvokes the explore task. Eventually, the desired knowledge is discovered, noticed by matching the exploration goal against the state, and the exploration task returns its augmented world knowledge. At this point, the robot again attempts to form a plan to achieve its goal. However, the plan is computed from the current situation, with the robot at the room in which it discovered the wrench. With the knowledge of the wrench, the plan is built successfully and asserted within the solve task.

Finally, the presence of the plan that achieves the desired goal causes a new task to be created, that of executing the plan. The execute task checks each step of the plan against the actual world state. No discrepancies are found so the plan is executed successfully. This causes the execution and solve tasks to be completed, and the select task is popped to consider the next goal.

To summarize, the main point of this example is the rational interleaving of planning, execution, and learning. The robot first attempts to plan. After planning fails, the robot performs some directed physical actions to obtain more knowledge. While pursuing this task, the robot is interrupted by a more pressing task. Once the emergency is dealt with, the robot returns to its explorations. With the additional knowledge found by exploring, the robot successfully builds a plan. Once the plan is completed, it is executed.

6. Summary

This paper has presented a general architecture that supports explicit reasoning about cognitive capabilities. The power of the architecture is derived from a knowledge representation that provides a data structure appropriate to reasoning, and a multi-level control structure that yields reactive knowledge-based behavior. Both components emphasize modularity and the use of abstractions as a practical requirement for implementing large systems.

In addition, a collection of knowledge was described that implements the integration of planning, execution, and learning. Among the bodies of knowledge are the problem solving domain that intelligently applies other capabilities, and several intentional learning paradigms that allow the system to actively seek knowledge based on need.

A working implementation of the MAX architecture has been completed, and the key elements of the knowledge used in the example are nearly encoded. In addition, a simple simulator has been written to update the environment and provide sensory data. The remaining work will focus on completing and augmenting the capability domains to allow more complex scenarios to be handled.

References

[1] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the National Conference on Artificial Intelligence*. 1987.

[2] J. R. Anderson. *The Architecture of Cognition*. Harvard University Press, 1983.

[3] Rodney Brooks. *A Robust Layered Control System for a Mobile Robot*. Technical Report 864, Artificial Intelligence Laborabory, Massachusetts Institute of Technology, 1985.

[4] J.G. Carbonell and Yolanda Gil. Learning by experimentation. In *Proceedings of the Forth International Workshop on Machine Learning*. 1987.

[5] Michael R. Genesereth. An Overview of Meta-Level Architecture. In *Proceedings of the Third Annual National Conference on Artificial Intelligence*, pages 119-124. 1983.

[6] Michael P. Georgeff, Amy L. Lansky, and Marcel J. Schoppers. *Reasoning and Planning in Dynamic Domains: An Experiment With a Mobile Robot*. Technical Report 380, Artificial Intelligence Center, SRI International, 1987.

[7] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence* 26, 1985.

[8] L. Kaelbling. An architecture for intelligent reactive systems. In M. Georgeff and A. Lansky (editors), *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*. 1987.

[9] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An Architecture for General Intelligence. *Artificial Intelligence* 33(1), 1987.

[10] Douglas B. Lenat. EURISKO: A program that learns new heuristics and domain concepts. The nature of heuristics III: Program design and results. *Artificial Intelligence* 21(1 & 2), 1983.

[11] S. Minton, J. Carbonell, C. Knoblock, D. Kuokka, O. Etzioni, Y. Gil. Explanation-based learning: A problem-solving perspective. to appear in *Artificial Intelligence*, 1989.

[12] Tom M. Mitchell, John Allen, Prasad Chalasani, John Cheng, Oren Etzioni, Marc Ringuette, Jeff Schlimmer. Theo: A framework for self-improving systems. *Architectures for Intelligence*. Lawrence Earlbaum, 1989, in press.

[13] E.D. Sacerdoti. *A Structure for Plans and Behavior*. Elsevier, 1977.

[14] M. Stefik. Planning and Meta-Planning. *Readings in Artificial Intelligence*. Tioga, 1981.

[15] Anthony Stentz and Yoshimasa Goto. The CMU navigational architecture. In *Proceedings of the DARPA Image Understanding Workshop*. 1987.

[16] David E. Wilkins. *High-Level Planning in a Mobile Robot Domain*. Technical Report 388, Artificial Intelligence Center, SRI International, 1986.